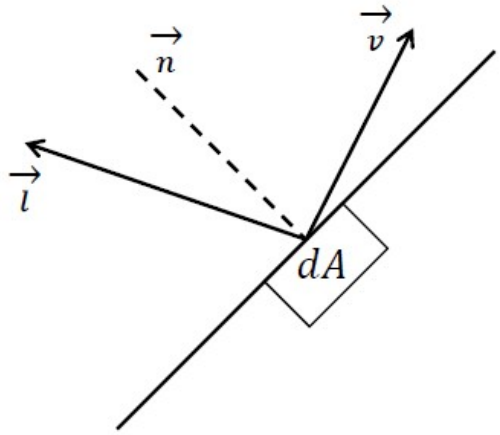
Computer Vision (16-720 B): Homework 6

Photometric Stereo

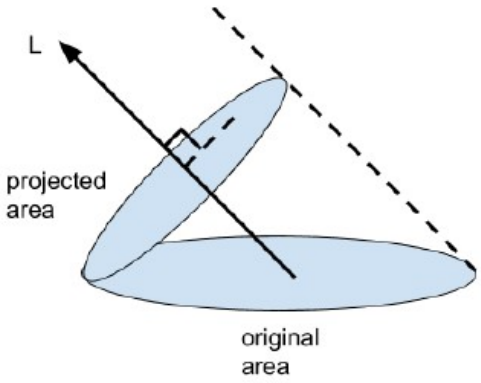
Name - Saharsh Agarwal
AndrewID - saharsh2

1 Calibrated Photometric Stereo

A. Understanding n-dot-l lighting



(a) Geometry of photometric stereo



(b) Projected area

In figure (a) above, $\cos\theta_i$ is the cosine of the angle of incidence(θ_i) between $\vec{l}$ and $\vec{n}$. Both $\vec{l}$ and $\vec{n}$ are unit vectors and thus,

$$\cos\theta_i = \vec{l}.\vec{n} \tag{1}$$

For the n-dot-l lighting, Surface Radiance (L) and Illumination of the surface (E) is related by -

$$L = \frac{\rho_d}{\pi}E = \frac{\rho_d}{\pi}\frac{J}{r^2}(\vec{l}.\vec{n}) \tag{2}$$

where J is the brightness of the source. J depends on the solid angle subtended, which in turns depend on the projected area, thereby introducing the term (and $\cos\theta_i$).

$$OriginalArea \times \cos\theta_i = ProjectedArea \tag{3}$$

Lambertian Surface - surface appears equally bright in all directions as light diffuses in all directions. Thus, the viewing direction doesn't matter.

B. Rendering n-dot-l lighting

Figure 1: Rendering images in different lightings

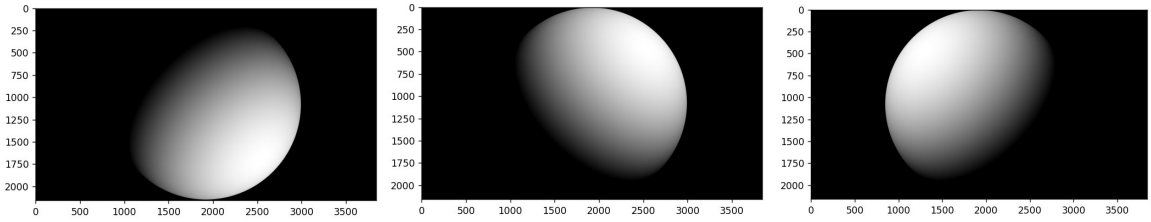


Figure 2: Rendering images in different lightings - cos theta only (allowing -ve values)

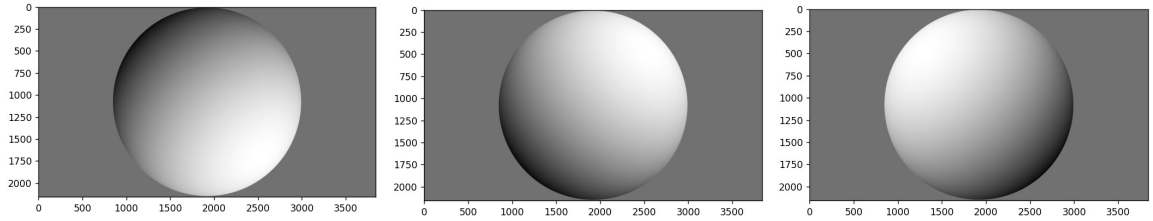


Figure 3: Code for **renderNDotLSphere**

```
image : numpy.ndarray
    The rendered image of the hemispherical bowl
"""

### print(center, rad, light, pxSize, res)
[X, Y] = np.meshgrid(np.arange(res[0]), np.arange(res[1]))
# print(X.shape, Y.shape) #2160, 3840
X = (X - res[0]/2) * pxSize*1.e-4 # everything in cm
Y = (Y - res[1]/2) * pxSize*1.e-4
Z = np.sqrt(rad**2+0j-X**2-Y**2)
X[np.real(Z) == 0] = 0
Y[np.real(Z) == 0] = 0
Z = np.real(Z)
# print(Z[1080,1920])
# plt.imshow(Z)
# plt.show() ## sphere

x = X.reshape(-1)
y = Y.reshape(-1)
z = Z.reshape(-1)
pts = np.vstack((np.vstack((x,y)),z))

dist = np.linalg.norm(pts,axis=0)+1.e-8 #to avoid 0/0

# print(dist[36450])
# norm_pts = pts
# print(norm_pts.shape)
# for i in range(3):
#     print(i, norm_pts[i].shape, pts[i].shape,dist.shape)
#     norm_pts[i] = pts[i]/dist #could have just divided the image by 0.75
#print(norm_pts.shape,light.shape)

n_pts = pts/dist
image = np.dot(n_pts.T,light).reshape((res[1],res[0]))

#print(image[1920])

a = image<0
image[a]=0
return image
```

C. Loading Data

Figure 4: Code for **loadData**

```
# loadData function
I = None
L = None
s = None

for i in range(7):
    pth = os.path.join(path,"input_" + str(i+1) + ".tif")
    print(pth)
    img = skimage.img_as_uint(skimage.io.imread(pth))
    #print(img.shape) #431,369,3
    c_img = rgb2xyz(img)
    l_img = c_img[:,:,1] #Y channel for luminance
    if I is None:
        pcount = l_img.shape[0]*l_img.shape[1]
        I = np.zeros((7,pcount))
        s = l_img.shape
    I[i,:] = l_img.reshape((1,pcount)) #P=pcount

L = np.load(os.path.join(path,"sources.npy")).T
print(L.shape)

return I, L, s
```

D. Initials

Upon performing Singular Value Decomposition on I, the singular values found:
79.36348099, 13.16260675, 9.22148403, 2.414729, 1.61659626, 1.26289066, and 0.89368302.

Since, $\vec{n}$ has 3 values being in the 3D coordinates. Thus, it requires lighting from 3 linearly independent directions to determine it completely. Hence, Intensity Matrix should have a rank equal to 3 ideally.

However, the singular values achieved upon SVD of I $\implies$ suggest a rank of greater than 3 (rank of our I matrix is 7). It doesn't agree with our Rank-3 requirements.

Figure 5: Code for **SVD** of I

```
# Part 1(d)
u,sigma,vh = np.linalg.svd(I,full_matrices=False)
print(sigma)
```

As we know, the world is not ideal and so are the image capturing and rendering processes. Image captured might have irregularities because of which the rank exceeds 3. More images from multiple light sources & directions, will better help with image reconstruction.

E. Estimating Pseudonormals

$$L^T B = I \implies B = (L^T)^{-1} I \tag{4}$$

Ax=y is converted to $x = A^{-1}y$ in the above equation. Since L is not square, L inverse is difficult to calculate. Thus,

$$B = (L^T)^{-1} I \implies B = (L^T)^{-1}(L^{-1}L)I \tag{5}$$

$$B = ((L^T)^{-1}L^{-1})LI \implies B = (LL^T)^{-1}LI \tag{6}$$

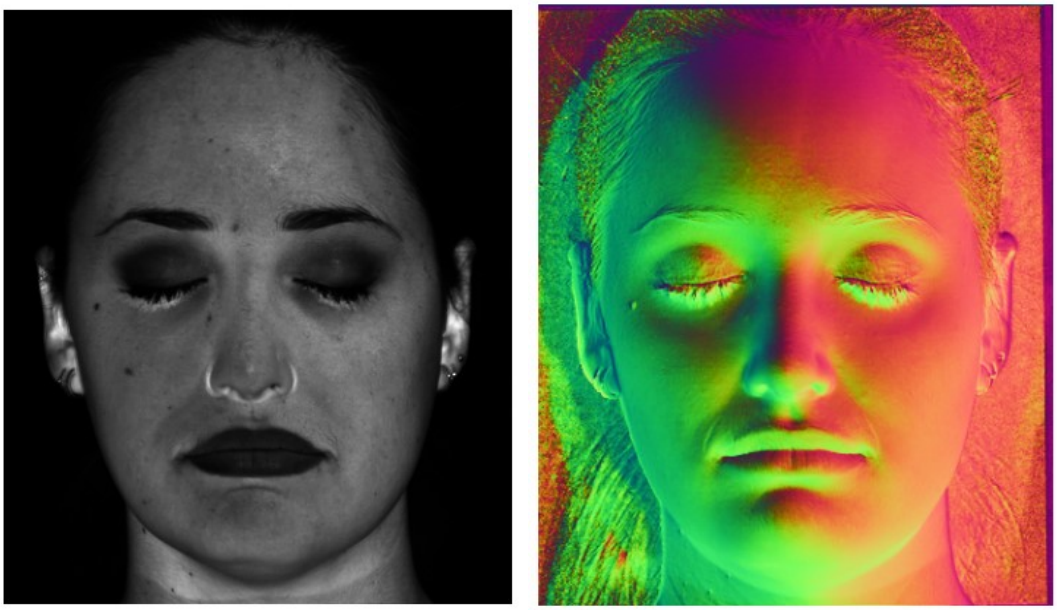
Now, LL^T is square matrix and hence, inverse is possible. So, A = LL^T and y = LI.

Figure 6: Code for **estimatePseudonormalsCalibrated**

```
def estimatePseudonormalsCalibrated(I, L):

    """
    Question 1 (e)

    In calibrated photometric stereo, estimate pseudonormals from the
    light direction and image matrices

    Parameters
    ----------
    I : numpy.ndarray
        The 7 x P array of vectorized images

    L : numpy.ndarray
        The 3 x 7 array of lighting directions

    Returns
    -------
    B : numpy.ndarray
        The 3 x P matrix of pesudonormals
    """

    B = np.linalg.inv(L@L.T)@L@I
    #print(B.shape) #3,159039 (431*369)
    return B
```

F. Albedos and Normals

Figure 7: Calibrated photometric stereo Results



The gray image was obtained by clipping the brightness as very few pixels have very high value. The normals and the curvature of the face is obtained as expected. However, when I analysed the real images it was clear that in some pictures the ears were either partially or completely invisible in the darkness. Hence, the reconstruction is not impeccable. Also, we see brighter spots in the gray image - near the ears and nose. I believe the eclipsing in the darkness (due to shadowing) will be the cause for it.

Figure 8: Code for **estimateAlbedosNormals**

```
def estimateAlbedosNormals(B):

    '''
    Question 1 (e)

    From the estimated pseudonormals, estimate the albedos and normals

    Parameters
    ----------
    B : numpy.ndarray
        The 3 x P matrix of estimated pseudonormals

    Returns
    -------
    albedos : numpy.ndarray
        The vector of albedos

    normals : numpy.ndarray
        The 3 x P matrix of normals
    '''

    albedos = np.linalg.norm(B, axis=0)
    #print(albedos.shape)#all the pixels - pcount
    normals = B/(albedos+1.e-8)
    return albedos, normals
```

Figure 9: Code for **displayAlbedosNormals**

```
def displayAlbedosNormals(albedos, normals, s):

    """
    Question 1 (f, g)

    From the estimated pseudonormals, display the albedo and normal maps

    Please make sure to use the `coolwarm` colormap for the albedo image
    and the `rainbow` colormap for the normals.

    Parameters
    ----------
    albedos : numpy.ndarray
        The vector of albedos

    normals : numpy.ndarray
        The 3 x P matrix of normals

    s : tuple
        Image shape

    Returns
    -------
    albedoIm : numpy.ndarray
        Albedo image of shape s

    normalIm : numpy.ndarray
        Normals reshaped as an s x 3 image

    """
    mina = np.min(albedos)
    maxa = np.max(albedos)
    al_norm = (albedos-mina)/(maxa-mina)
    albedoIm = al_norm.reshape(s)
    # print(normals.shape) 3,pcount
    normalIm = (((normals+1)/2).T).reshape((s[0],s[1],3))

    minn = np.min(normalIm)
    maxn = np.max(normalIm)
    normalIm = (normalIm-minn)/(maxn-minn)

    return albedoIm, normalIm
```

G. Normals and Depth

Assume the real surface to be smooth at and around P(x,y) image point - to get normal $\vec{n}$ or (n1,n2,n3). We will analyse points P1(x,y+1) and P2(x+1,y) in the image, projected by a scalar factor **s** in the real world.

V1 - vector between P and P1 in the 3D world.

V2 - vector between P and P2 in the 3D world.

$$V1 = (cx, c(y+1), Z_{x,y+1}) - (cx, cy, Z_{x,y}) = (0, c, Z_{x,y+1} - Z_{x,y}) \tag{7}$$

$$V2 = (c(x+1), cy, Z_{x+1,y}) - (cx, cy, Z_{x,y}) = (c, 0, Z_{x+1,y} - Z_{x,y}) \tag{8}$$

Assuming smooth differential surface for infinitesimal small change in the x or y direction - Normal to the surface ($\vec{n}$) will be perpendicular to the vectors V1 and V2.

$$\vec{n}.(V1) = 0 = (n1, n2, n3).(0, c, Z_{x,y+1} - Z_{x,y}) = n2(c) + n3(Z_{x,y+1} - Z_{x,y}) \tag{9}$$

$$\vec{n}.(V2) = 0 = (n1, n2, n3).(c, 0, Z_{x+1,y} - Z_{x,y}) = n1(c) + n3(Z_{x+1,y} - Z_{x,y}) \tag{10}$$

$z_{x,y}$ = f(x,y). Differentiating equations 9 and 10 at (x,y), such that delta change is c-

$$\frac{dz}{dy} = -\frac{n2}{n3} \tag{11}$$

$$\frac{dz}{dx} = -\frac{n1}{n3} \tag{12}$$

H. Understanding Integrability of Gradients

$$g_x = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, g_y = \begin{bmatrix} 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \end{bmatrix} \tag{13}$$

Yes, calculating it any of the 2 ways suggested yields the same result for finding gradients.

When we reconstruct g from the gradients g_x and g_y, we can integrate the latter 2 from a reference point P1 to any point P2 in the image. However, images have noise and there can be noise while calculating gradients at different points. So when we try to ingrate from one point to another, the integration will not be path independent like they should be. This will give different results based on what path we take on the image to integrate - thereby producing varying results for z. This makes the integration difficult.

To enforce integrability we insert a step to guarantee that when we integrate gradients, we will obtain the surface Z. 1 way is to minimise the calculate error between the measured gradients and the gradients recalculated after surface reconstruction. We also use Fourier transform and enforce integrability constraints - to make g_x and g_y integrable.

I. Shape Estimation

Figure 10: Code for **estimateShape** and Results

```
def estimateShape(normals, s):

    """
    Question 1 (j)

    Integrate the estimated normals to get an estimate of the depth map
    of the surface.

    Parameters
    ----------
    normals : numpy.ndarray
        The 3 x P matrix of normals

    s : tuple
        Image shape

    Returns
    ----------
    surface: numpy.ndarray
        The image, of size s, of estimated depths at each point

    """
    print(normals.shape)

    z_x = -(normals[0,:]/(normals[2,:]+1.e-8)).reshape(s)
    z_y = -(normals[1,:]/(normals[2,:]+1.e-8)).reshape(s)
    surface = integrateFrankot(z_x,z_y)
    #print(surface.shape)
    return surface
```

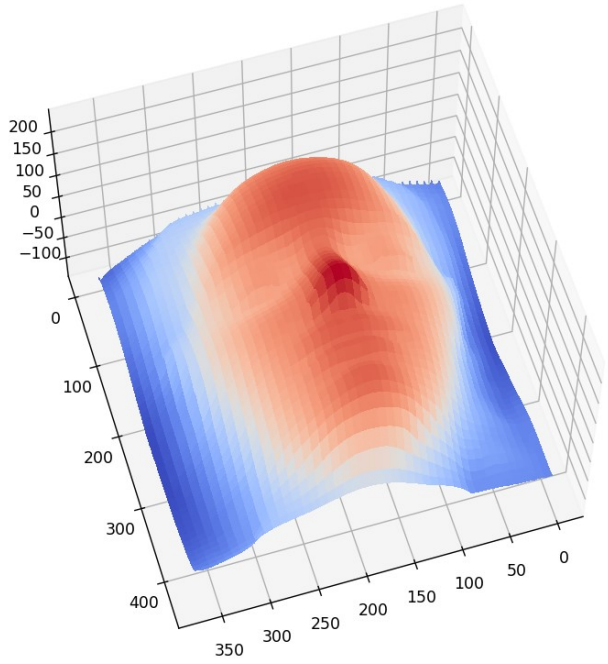
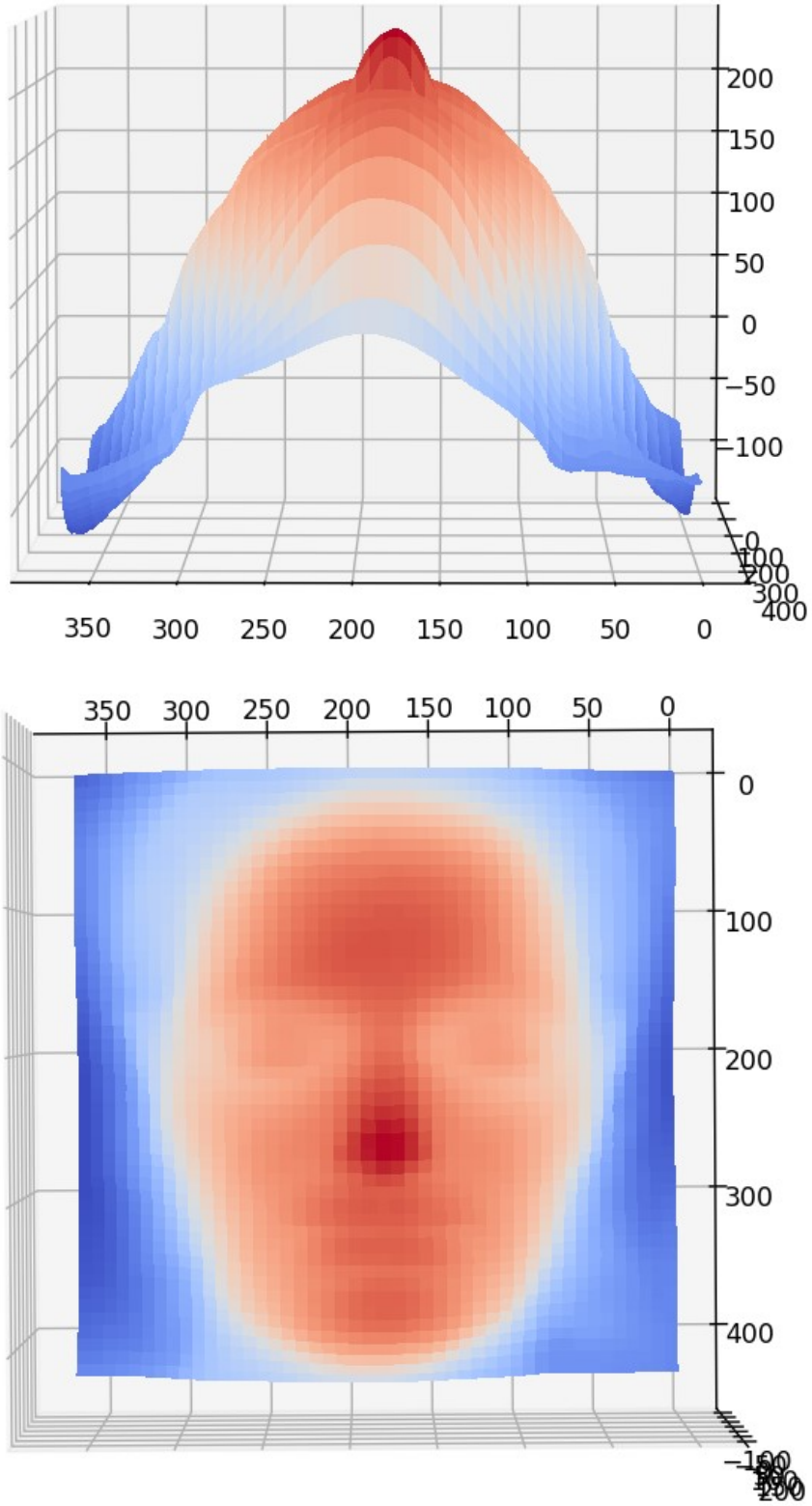


Figure 11: Results after **estimateShape**



2. Uncalibrated Photometric Stereo

A. Uncalibrated Normal Estimation

We perform SVD on the matrix I (dim = 7×P). This is done to get Light and scaled surface normals.

$$I = U\Sigma V^T \tag{14}$$

where U (left singular matrix) is 7x7 and V^T (right matrix) is PxP. We keep only the top 3 highest and important singular values from Σ and corresponding vectors in U and V^T. These vectors will be representative of L and B respectively. Additional constraints are put to make it integrable and get values as close to I as possible. This is done by adding identity matrix (AA^{-1}) between L-svd (estimated from U) and B-svd (estimated from V). This, L = (L-est).(A) and B = (A^{-1}).(B-est). Integrable surface normals B is then used to find Z (depth).

B. Calculation and Visualization

Figure 12: Uncalibrated Photometric Stereo - Albedos and Normals

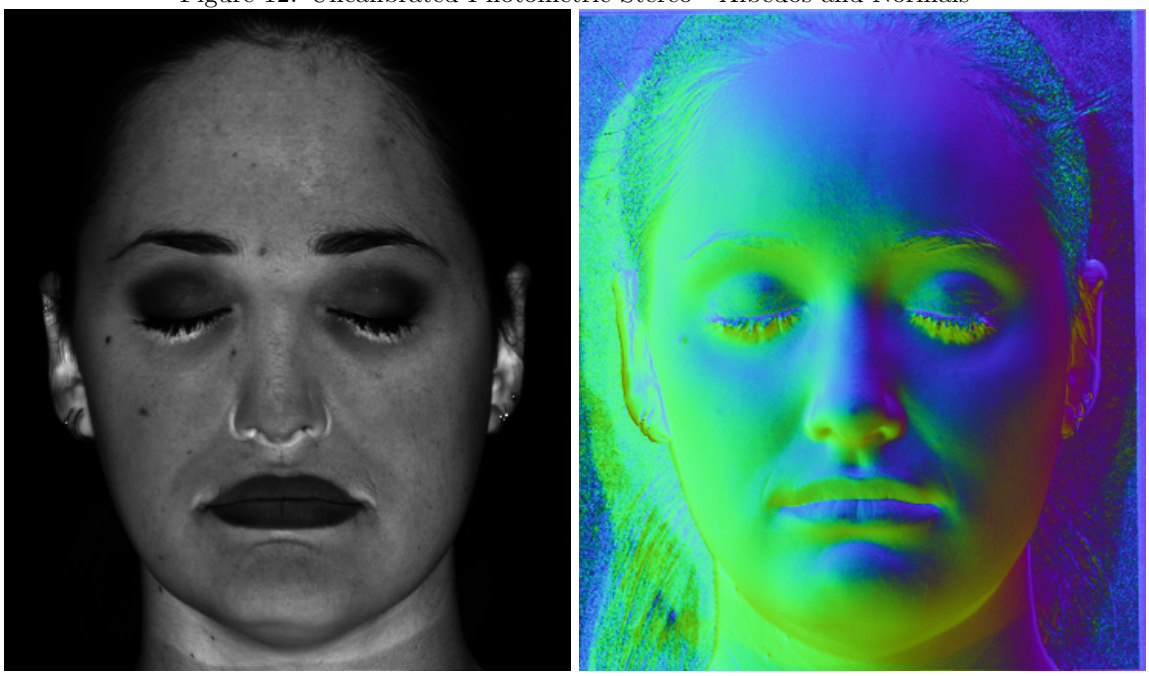


Figure 13: Code for **estimatePseudonormalsUncalibrated**

```
def estimatePseudonormalsUncalibrated(I):

    """
    Question 2 (b)

    Estimate pseudonormals without the help of light source directions.

    Parameters
    ----------
    I : numpy.ndarray
        The 7 x P matrix of loaded images

    Returns
    -------
    B : numpy.ndarray
        The 3 x P matrix of pseudonormals

    L : numpy.ndarray
        The 3 x 7 array of lighting directions

    """

    u, sigma, vh = np.linalg.svd(I,full_matrices=False)
    sigma[3:]=0
    B = vh[:3,:]
    L = u[:,:3]
    return B, L
```

C. Comparing to ground truth lighting

Original Lighting - (7 sources along columns)

$$\begin{bmatrix} -0.1418 & 0.1215 & -0.069 & 0.067 & -0.1627 & 0. & 0.1478 \\ -0.1804 & -0.2026 & -0.0345 & -0.0402 & 0.122 & 0.1194 & 0.1209 \\ -0.9267 & -0.9717 & -0.838 & -0.9772 & -0.979 & -0.9648 & -0.9713 \end{bmatrix}$$

Lighting Sources Estimated - (7 sources along columns)

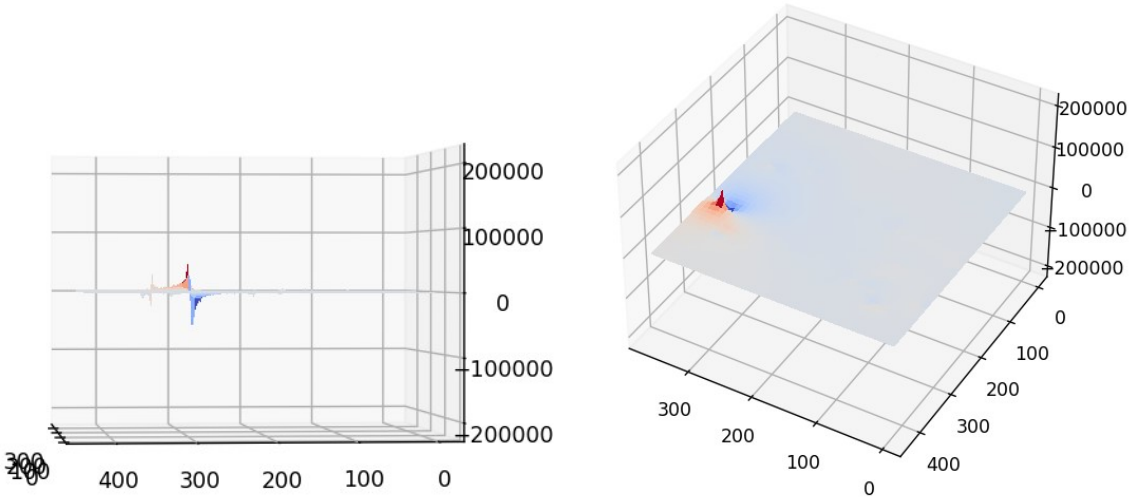
$$\begin{bmatrix} -0.33593029 & -0.43440914 & -0.27030342 & -0.42038035 & -0.4031327 & -0.38015618 & -0.37632601 \\ 0.26124499 & -0.63866314 & 0.13757076 & -0.17254382 & 0.64103139 & 0.1284584 & -0.21849616 \\ 0.61888016 & 0.3341193 & 0.14141256 & -0.00569798 & -0.10233937 & -0.30056934 & -0.62008579 \end{bmatrix}$$

The ground truth (original Lighting) is very different from the L (sources) we estimated.

Reading the bas-relief ambiguity, multiplying estimated B with a G matrix (given G of the form $= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \mu & \gamma & \lambda \end{bmatrix}$ will produce same results. Albedos and Normals are already normalised in display Fucntion.

D. Reconstructing the Shape : Attempt 1

Figure 14: Attempt 1 Results



The plots look nothing like the face in the pictures.

E. Reconstructing the Shape : Attempt 1 (Enforcing Integrability

Figure 15: Attempt 2 Results

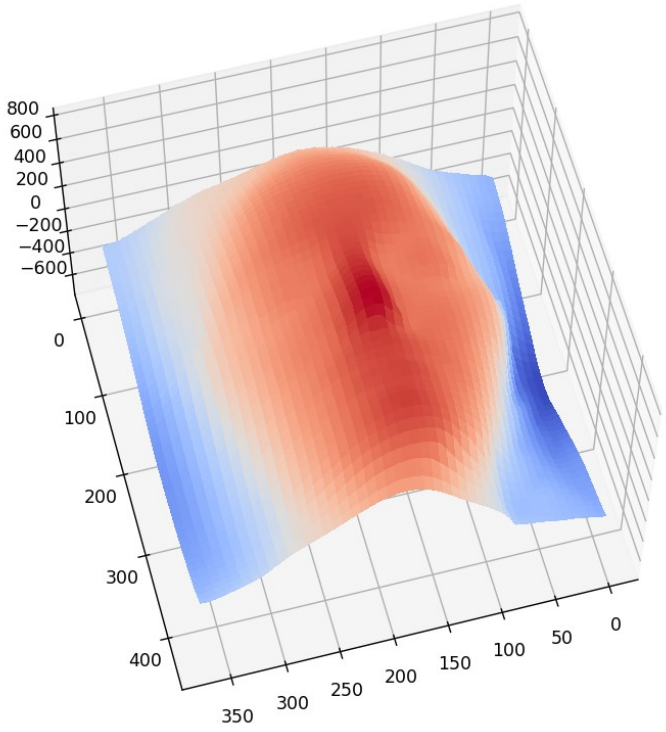
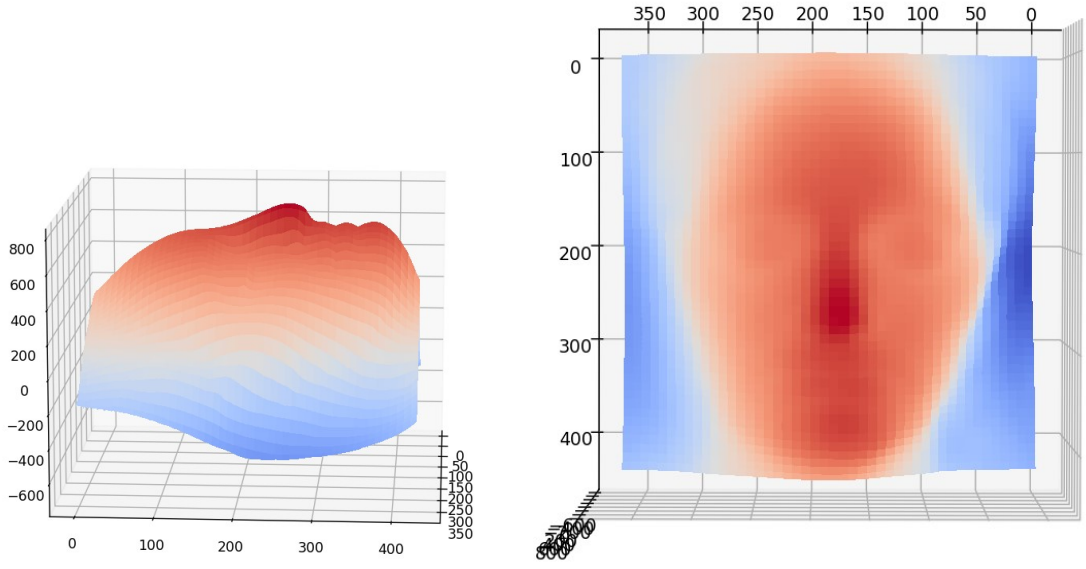


Figure 16: Attempt 2 Results



The images look very similar to calibrated images, hence, introducing the enforcement of integrability produces far better results. The look just like the ones expected in the writeup.

F. Why low relief?

Changing μ - 0.25, 1 and 2.

Figure 17: $\mu = 1, \gamma = 1, \lambda = 1$

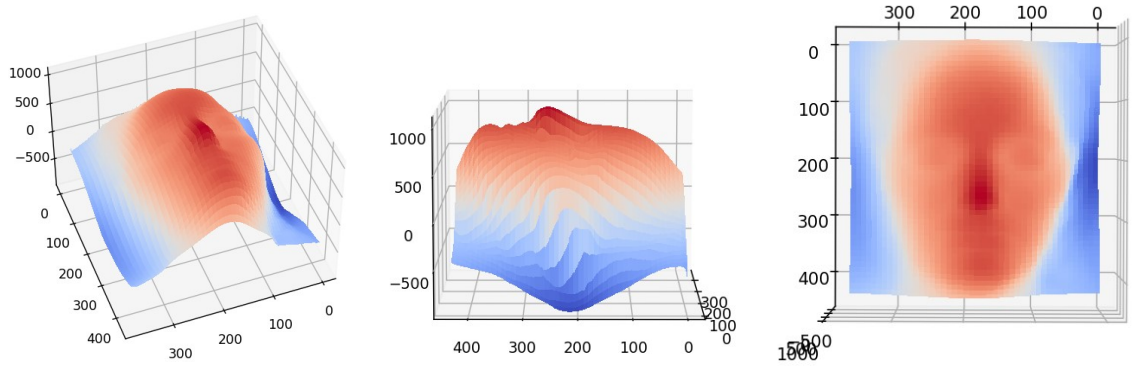


Figure 18: $\mu = 0.25, \gamma = 1, \lambda = 1$

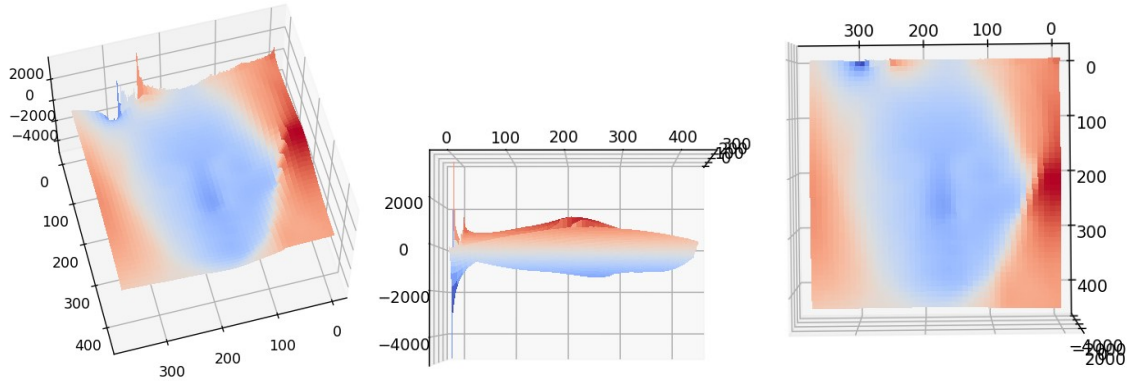
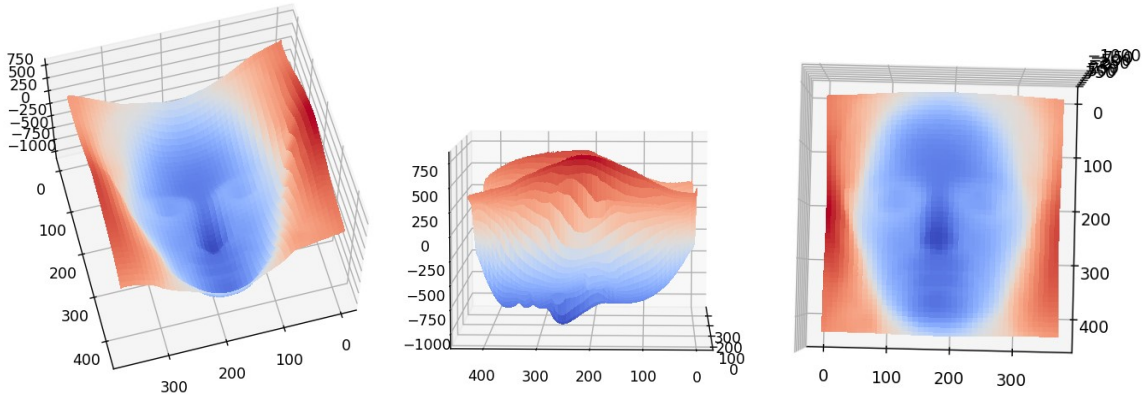


Figure 19: $\mu = 2, \gamma = 1, \lambda = 1$



Changing the μ from 1 to 2 inverts the image inside out. However, at or below μ =0.25 (cet par), the image formation just feels like it is beginning. Reducing it, reduces the height making it flatter.

Changing γ - 0.65, 1 and 100

Figure 20: $\mu = 1, \gamma = 1, \lambda = 1$

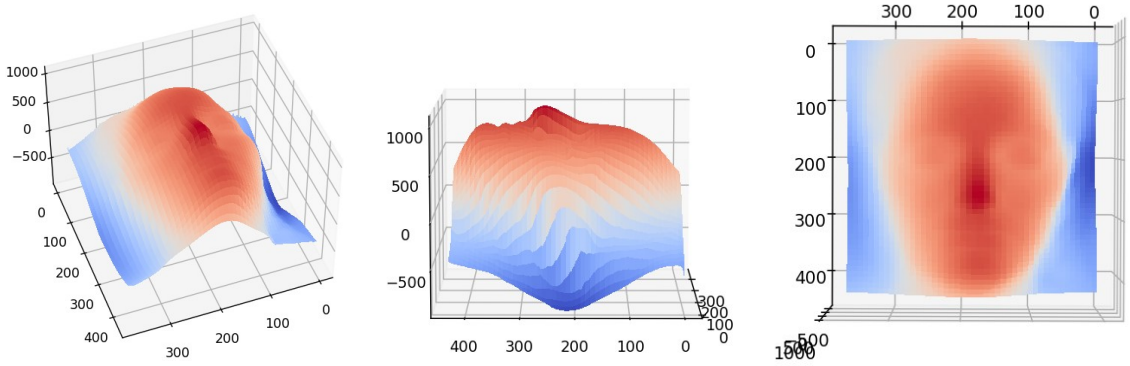


Figure 21: $\mu = 1, \gamma = 100, \lambda = 1$

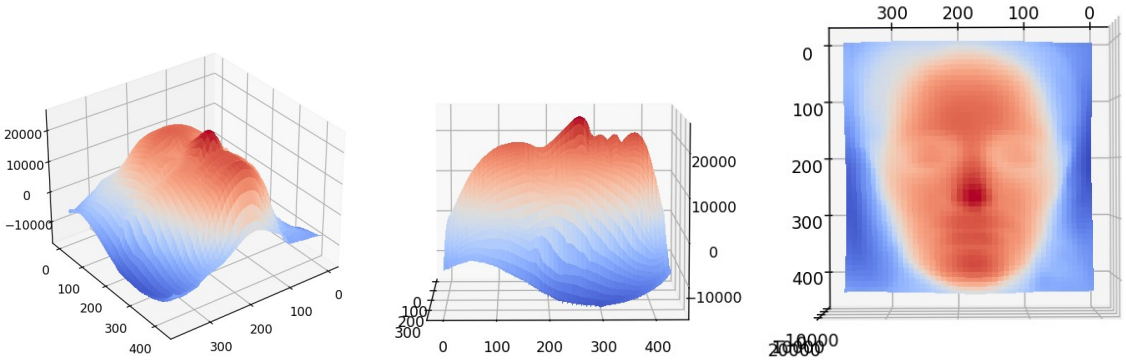
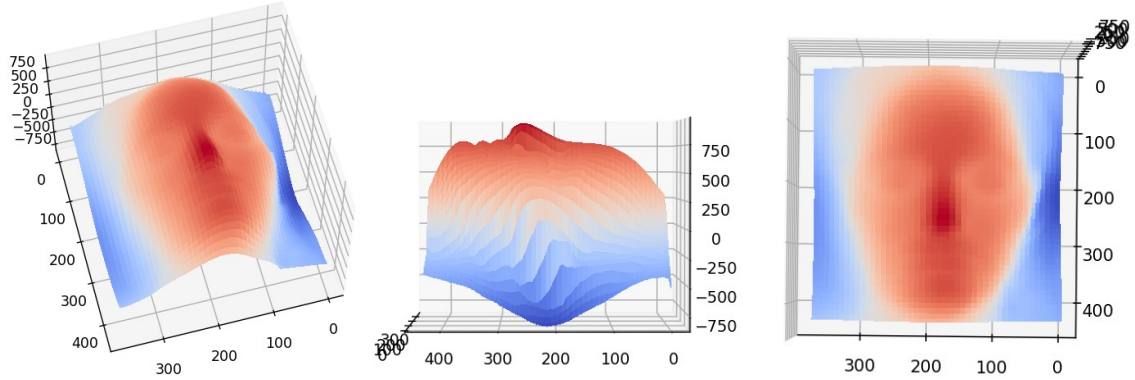


Figure 22: $\mu = 1, \gamma = 0.65, \lambda = 1$



Changing the γ from 1 to 100 gives sharper image - this is because of the elevated Z as seen from the graphs. Similarly, lowering it to 0.65 gives flatter view. However, below this, the image inverts inside-out. Reducing it makes the surface flatter.

Changing λ - 1, 5 and 15

Figure 23: $\mu = 1, \gamma = 1, \lambda = 1$

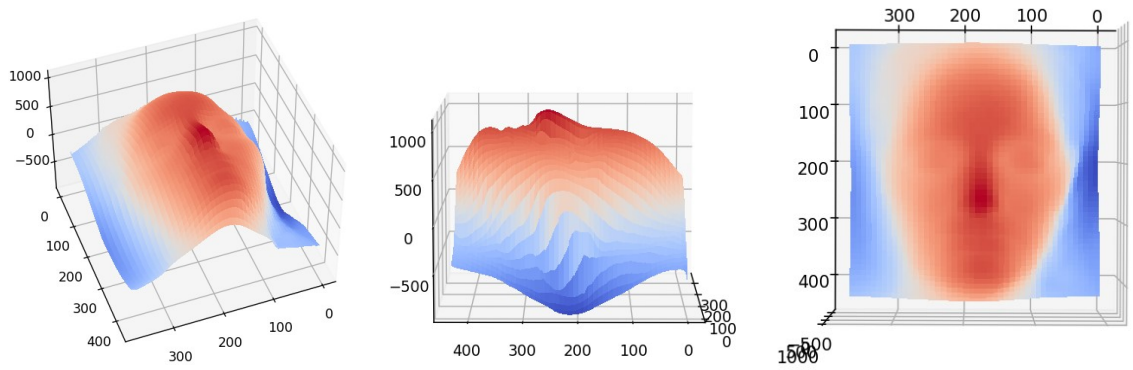


Figure 24: $\mu = 1, \gamma = 1, \lambda = 5$

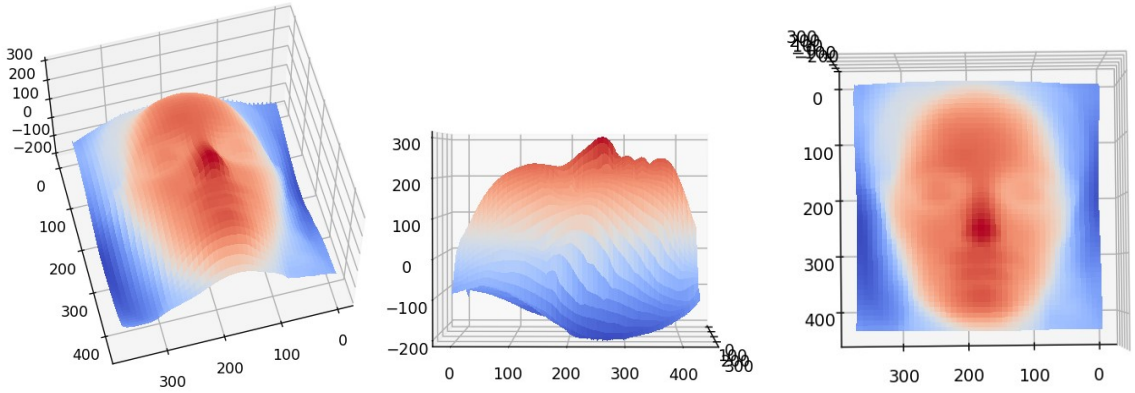
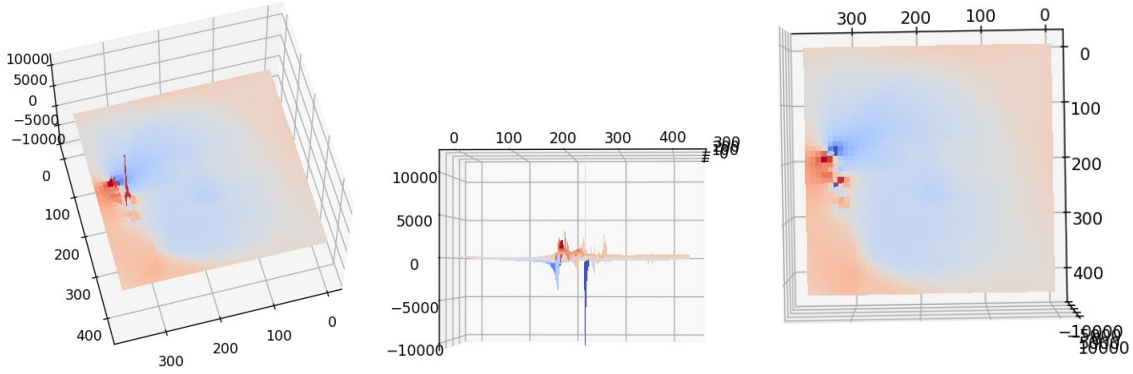


Figure 25: $\mu = 1, \gamma = 1, \lambda = 15$



Changing the λ from 1 to 5 gives a much flatter surface of the face. However, decreasing it close to 0 gives very sharp images due to increase in Z values. Taking λ to 15, gives a very flat image, with abrupt spikes around face edge.

Figure 26: Code for **plotBasRelief**

```
def plotBasRelief(B, mu, nu, lam):

    """
    Question 2 (f)

    Make a 3D plot of of a bas-relief transformation with the given parameters.

    Parameters
    ----------
    B : numpy.ndarray
        The 3 x P matrix of pseudonormals

    mu : float
        bas-relief parameter

    nu : float
        bas-relief parameter

    lambda : float
        bas-relief parameter

    Returns
    -------
        None

    """
    G = np.asarray([[1,0,0],[0,1,0],[mu,nu,lam]])
    #print(G)
    new_B_est = np.linalg.inv(G.T) @ B_est
    #when inverted
    # x = np.eye(3)
    # x[-1,-1] = -1
    # new_B_est = np.linalg.inv(x) @ new_B_est

    albedos_est, normals_est = estimateAlbedosNormals(new_B_est)
    normals_est = enforceIntegrability(normals_est,s)
    albedoIm_est, normalIm_est = displayAlbedosNormals(albedos_est, normals_est, s)

    surface = estimateShape(normals_est, s)
    plotSurface(surface)
```

Bas-relief ambiguity is called so because of the ambiguity in determining the 3D shape of an object just by illuminated pictures (with shadows). The information is not enough for completely finding the Z points. As we have seen, changing G's variables will alter the shape of the face.

G. Flattest Surface Possible

As clear from previous pictures, increasing λ helps flattening the image. Hence, making λ as large as possible helps.

Similarly, from the previous section - reducing μ and γ produces flat results. Hence, μ and γ are reduced as low as possible following the constraints of being non-negative.

Thus, for the flattest surface - put μ and γ as 0 and λ as very high positive number (c). Thus, G $= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & c \end{bmatrix}$

H. More Measurements

Acquiring more pictures from more lighting directions will provide more information about the picture like shadows, etc. Hence, it can be better for resolving ambiguity - by minimising errors and noise. If lighting varies heavily, they will help better.